Perceptrons

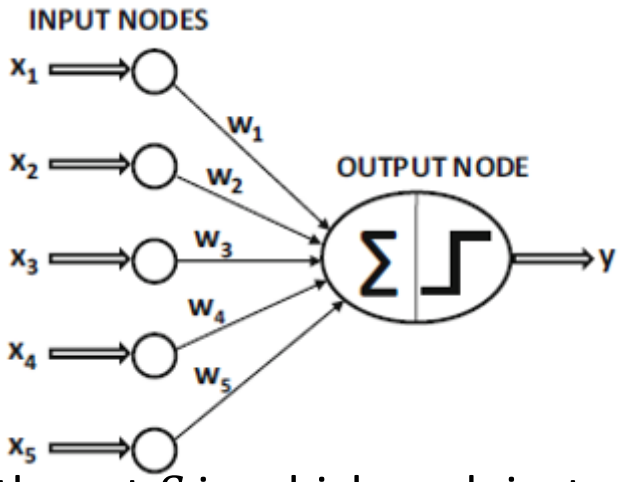
AW

Lecture Overview

1. Basic Architecture - Perceptron

2. Activation and Loss functions

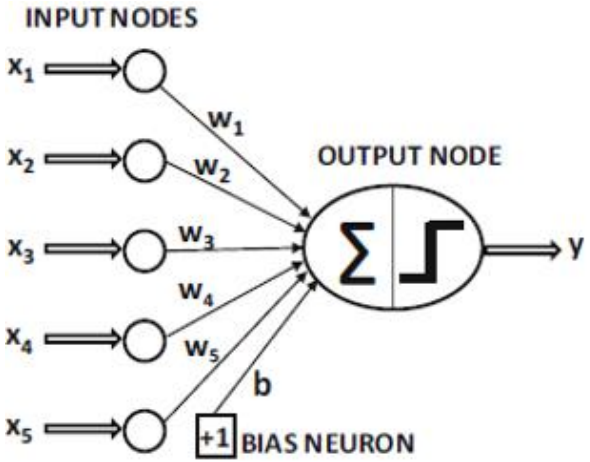
Single Layer: Perceptron, the Device



- Assume training the set S in which each instance the is of the form $(\vec{x}, y)$, where $\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix}$ is the feature vector and $y \in \{-1,1\}$
- The input layer contains d nodes that transmit the d features $x_1, \ldots, x_d$ with edges of weight $\vec{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix}$ to an output node
- The output layer computes $\cdot$ -product $\vec{w} \cdot \vec{x}$ = linear function $\sum_{i=1}^{d} w_i \cdot x_i$ and outputs label $\hat{y}$ that is sign of the sum. That is

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{d} w_i \cdot x_i\right) \text{ where } \text{sign(x)} = \begin{cases} 1 \; if \; x > 0 \\ -1 \; if \; x \leq 0 \end{cases}$$

Perceptron with Bias



- As before each training instance is of the form $(\vec{x}, y)$ with $\vec{x}$ feature vector and $y \in \{-1,1\}$ is the class. The input layer contains transmit features (coordinates) $x_1, \ldots, x_d$ of $\vec{x}$ with edges of weight $\vec{w}$ to an output node.
- Constant *bias* b is an additional input to the output node
- The output layer computes $\vec{w} \cdot \vec{x} + b = \sum_{i=1}^{d} w_i \cdot x_i + b$ and outputs sign of this sum that is label $\hat{y}$, i.e. $\hat{y} = \text{sign}\left(\sum_{i=1}^{d} w_i \cdot x_i + b\right)$
- Formally reducible to perceptron without bias: let $\vec{w}' = \begin{pmatrix} b \\ \vec{w} \end{pmatrix}$ and $\vec{x}' = \begin{pmatrix} 1 \\ \vec{x} \end{pmatrix}$. Then $\hat{y} = \vec{w}' \cdot \vec{x}'$

Perceptron: Goal of Training

Let $S = \{(\vec{x}_i, y_i)\}_{i=1}^{n}$ be our training set, i.e. collection of points $\vec{x}$ wit known labels y

The goal: Train so that there is as little difference as possible between labels $\hat{y}$ that perceptron computes and true labels y.

Formally, find vector $\vec{w}^*$ such that

$$\begin{aligned}
\vec{w}^* &= \underset{\vec{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{(\vec{x},y)\in S} |y - \hat{y}| \\
&= \underset{\vec{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{(\vec{x},y)\in S} (y - \hat{y})^2 \\
&= \underset{\vec{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{(\vec{x},y)\in S} \left(y - \operatorname{sign}\left(\sum_{i=1}^{d} w_i \cdot x_i + b\right)\right)^2 \\
&= \underset{\vec{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{(\vec{x},y)\in S} \left(y - \operatorname{sign}\left(\begin{pmatrix} b \\ \vec{w} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vec{x} \end{pmatrix}\right)\right)^2 \\
&= \underset{\vec{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{(\vec{x},y)\in S} (y - \operatorname{sign}(\vec{w}' \cdot \vec{x}'))^2
\end{aligned}$$

Naïve Interpretation: Training Algorithm

- The training algorithm of neural networks works by feeding each input data instance $\vec{x}$ into the network (perceptron) one by one to create the prediction $\hat{y}$.
- The weights are then updated, based on the error value $E(\vec{x}) = (y - \hat{y})$, i.e. when the data point $\vec{x}$ is fed into the perceptron, the weight vector $\vec{w}$ is updated:

$$\vec{w}' \Leftarrow \vec{w}' + \alpha(y - \vec{y})\vec{x}' = \vec{w}' + \alpha E(\vec{x})\vec{x}'$$

- Parameter α regulates the learning rate of the perceptron
- The perceptron algorithm repeatedly cycles through all the training examples in random order and iteratively adjusts the weights until convergence is reached. One cycle over training data is called $epoch$.
- The algorithm terminates when the initial vector of an epoch have not changed (relatively previous epoch)
- Later: algorithm implicitly minimizes the squared error of prediction of smooth real valued function $y' = f(\vec{x})$ that coincided with staircase class function y in points $\vec{x} \in S$ given by training set S by performing gradient-descent updates on y'

Example: Perceptron learning of XOR not doable

XOR: $z = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$; $y = \text{sign}(z)$. Let learning rate be $\alpha = \frac{1}{2}$ and $b = 0$

Data: y and (z): Initialize:

Epoch 0 (before the start of the perceptron:

$\vec{x}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, y_1 = 1(1)$: $\vec{w}' = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$,

$\vec{x}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, y_2 = 1(1)$: ,

$\vec{x}_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, y_3 = -1(0)$: ,

$\vec{x}_4 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, y_4 = -1(0)$:

Example: Perceptron learning of XOR not doable

XOR: $z = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$; $y = \text{sign}(z)$. Let learning rate be $\alpha = \frac{1}{2}$ and $b = 0$

Data: y and (z): Initial vector: Step:

Epoch 1:

$\vec{x}_1 = \begin{pmatrix}1\\0\end{pmatrix}, y_1 = 1(1)$: $\vec{w}' = \begin{pmatrix}0\\0\\0\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_1' = 0(-) \neq y_1 \Rightarrow E(x) = 2\ update: \vec{w}' = \vec{w}' + \frac{1}{2} 2\vec{x}_1' = \begin{pmatrix}1\\1\\0\end{pmatrix}$

$\vec{x}_2 = \begin{pmatrix}0\\1\end{pmatrix}, y_2 = 1(1)$: $\vec{w}' = \begin{pmatrix}1\\1\\0\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_2' = 1\ (+) = y_2 \Rightarrow E(x) = 0\ no\ update$

$\vec{x}_3 = \begin{pmatrix}1\\1\end{pmatrix}, y_3 = -1(0)$: $\vec{w}' = \begin{pmatrix}1\\1\\0\end{pmatrix}, \vec{w}' \cdot \vec{x}_3' = 1 \neq y_3 \Rightarrow E(x) = -2\ update: \vec{w}' = \vec{w}' + \frac{1}{2}(-2)\vec{x}_3' = \begin{pmatrix}0\\0\\-1\end{pmatrix}$

$\vec{x}_4 = \begin{pmatrix}0\\0\end{pmatrix}, y_4 = -1(0): \vec{w}' = \begin{pmatrix}0\\0\\-1\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_4' = 0 = y_3 \Rightarrow E(x) = 0\ no\ update$

Example: Perceptron learning of XOR not doable

XOR: $z = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$; $y = \text{sign}(z)$. Let learning rate be $\alpha = \frac{1}{2}$ and $b = 0$

Data: y and (z): Initial vector: Step:

Epoch 2:

$\vec{x}_1 = \begin{pmatrix}1\\0\end{pmatrix}, y_1 = 1(1)$: $\vec{w}' = \begin{pmatrix}0\\0\\-1\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_1' = 0(-) \neq y_1 \Rightarrow E(x) = 2\ update: \vec{w}' = \vec{w}' + \frac{1}{2} 2\vec{x}_1' = \begin{pmatrix}1\\1\\-1\end{pmatrix}$

$\vec{x}_2 = \begin{pmatrix}0\\1\end{pmatrix}, y_2 = 1(1)$: $\vec{w}' = \begin{pmatrix}1\\1\\-1\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_2' = 0\,(-) \neq y_2 \Rightarrow E(x) = 2\ update: \vec{w}' = \vec{w}' + \frac{1}{2} 2\vec{x}_2' = \begin{pmatrix}2\\1\\0\end{pmatrix}$

$\vec{x}_3 = \begin{pmatrix}1\\1\end{pmatrix}, y_3 = -1(0): \vec{w}' = \begin{pmatrix}2\\1\\0\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_3' = 3(+1) \neq y_3 \Rightarrow E(x) = -2\ update: \vec{w}' = \vec{w}' + \frac{1}{2}(-2)\vec{x}_3' = \begin{pmatrix}1\\0\\-1\end{pmatrix}$

$\vec{x}_4 = \begin{pmatrix}0\\0\end{pmatrix}, y_4 = -1(0): \vec{w}' = \begin{pmatrix}1\\0\\-1\end{pmatrix}$, $\vec{w}' \cdot \vec{x}_4' = 1 \neq y_3 \Rightarrow E(x) = -2\ update: \vec{w}' = \vec{w}' + \frac{1}{2}(-2)\vec{x}_4' = \begin{pmatrix}0\\0\\-1\end{pmatrix}$

The weight vector after the second pass is the same as the weight vector after the first pass, so the algorithm will not change weight vectors – it converged. But values are not XOR!

Lecture Overview

1. Basic Architecture - Perceptron

2. Activation and Loss functions

3. Multilayer Networks

Perceptron's Loss

The number of classification errors in a binary classification problem for a training data set S can be computed as:

$$L^{0/1} = \frac{1}{2}\sum_{x_i \in S}\left(y_i - \text{sign}\left(\vec{w}' \cdot \vec{x}_i{}'\right)\right)^2 = \frac{1}{2}\sum_{\vec{x}_i \in S} 1 - y_i \text{sign}\left(\vec{w}' \cdot \vec{x}_i{}'\right)$$

It is not differentiable - has a staircase-like shape (from addition over multiple points). Derivative behaves as derivative of $\sum_{x_i} y_i \text{sign}\left(\vec{w}' \cdot \vec{x}_i{}'\right)$ in which additive terms are not differentiable because of the sign function.

Perceptron's Loss - Smooth Approximation

Later (again): updates of the perceptron implicitly optimize the *perceptron criterion* defined by dropping the sign and setting negative values to 0:

$$\arg\min_{\vec{w}} L_g(\vec{w}) = \arg\min_{\vec{w}} \sum_{x_i \in S} \max\left(0, -y_i\left(\vec{w}' \cdot \vec{x}_i{}'\right)\right)$$

- Correct predictions do not contribute to loss!
- The update rule of the perceptron is roughly

$$\vec{w} = \vec{w}' + \alpha \nabla_{\vec{w}} L(\vec{x}_i)$$

where $L(\vec{x}_i) = \max\left(0, -y_i\left(\vec{w}' \cdot \vec{x}_i{}'\right)\right)$ and $L_g(\vec{w}) = \sum_{x_i \in S} L(\vec{x}_i)$

- modified loss function that has the extremum at the same value of argument as an original non-differentiable loss function but allows for gradient computation is referred to as a *smoothed surrogate loss function*.

Surrogate Loss

- If a plane $[\vec{w}^*: b]$ that separates data set (i.e. there is a vector $\vec{w}^*$ such that $\vec{w}^* \cdot \vec{x}_i > b$ whenever $(\vec{x}_i, 1) \in S$ and $\vec{w}^* \cdot \vec{x}_j < b$ whenever $(\vec{x}_i, -1) \in S$ then $\min_{\vec{w}} L^{0/1}(\vec{w}) = \min_{\vec{w}} L_g(\vec{w}) = 0$
- Also

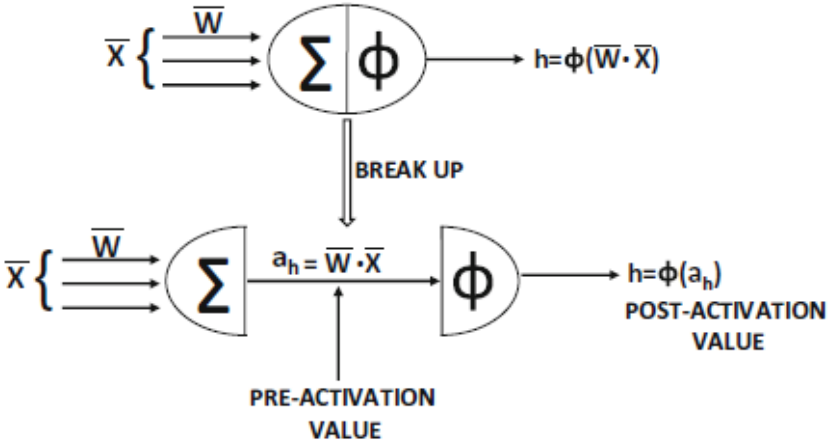
$$\vec{w}^* = \arg\min_{\vec{w}} L(\vec{w}) \Leftrightarrow \vec{w}^* = \arg\min_{\vec{w}} L_g(\vec{w})$$

- If data is linearly separable we can initialize perceptron training to $\vec{w}_0 = \vec{0}$ *irrespective of the training data set* in order to obtain the optimal loss value of 0.
- If data are not linearly separable then $\vec{w}^*$ returned by training algorithm is rather arbitrary
- Modified loss function (like L_g) that has the extremum at the same value of argument as an original non-differentiable loss function (e.g. $L^{0/1}$), but allows for gradient calculation of extremums is referred to as a *smoothed surrogate loss function*.

Activation Functions

- Neuron really computes two functions within the node:

$$\Phi(\vec{w} \cdot \vec{x}) = \Phi(\sum w_i x_i)$$



- Linear and different types of nonlinear functions such as the *sign*, *sigmoid*, or *hyperbolic tangents* may be used in various layers

Often Used Activation Functions

- The basic activation function is the identity $\Phi(v) = v$ which provides no nonlinearity:
  - It is actually linear activation as $v = \sum_{i=1}^{d} w_i x_i$.
- Popular activation functions:

$$\Phi(v) = \text{sign}(v)$$

$$\Phi(v) = \frac{1}{1+e^{-v}} = \frac{e^{v}}{e^{v}+1} = 1 - \Phi(-v) \quad \text{(sigmoid)}$$

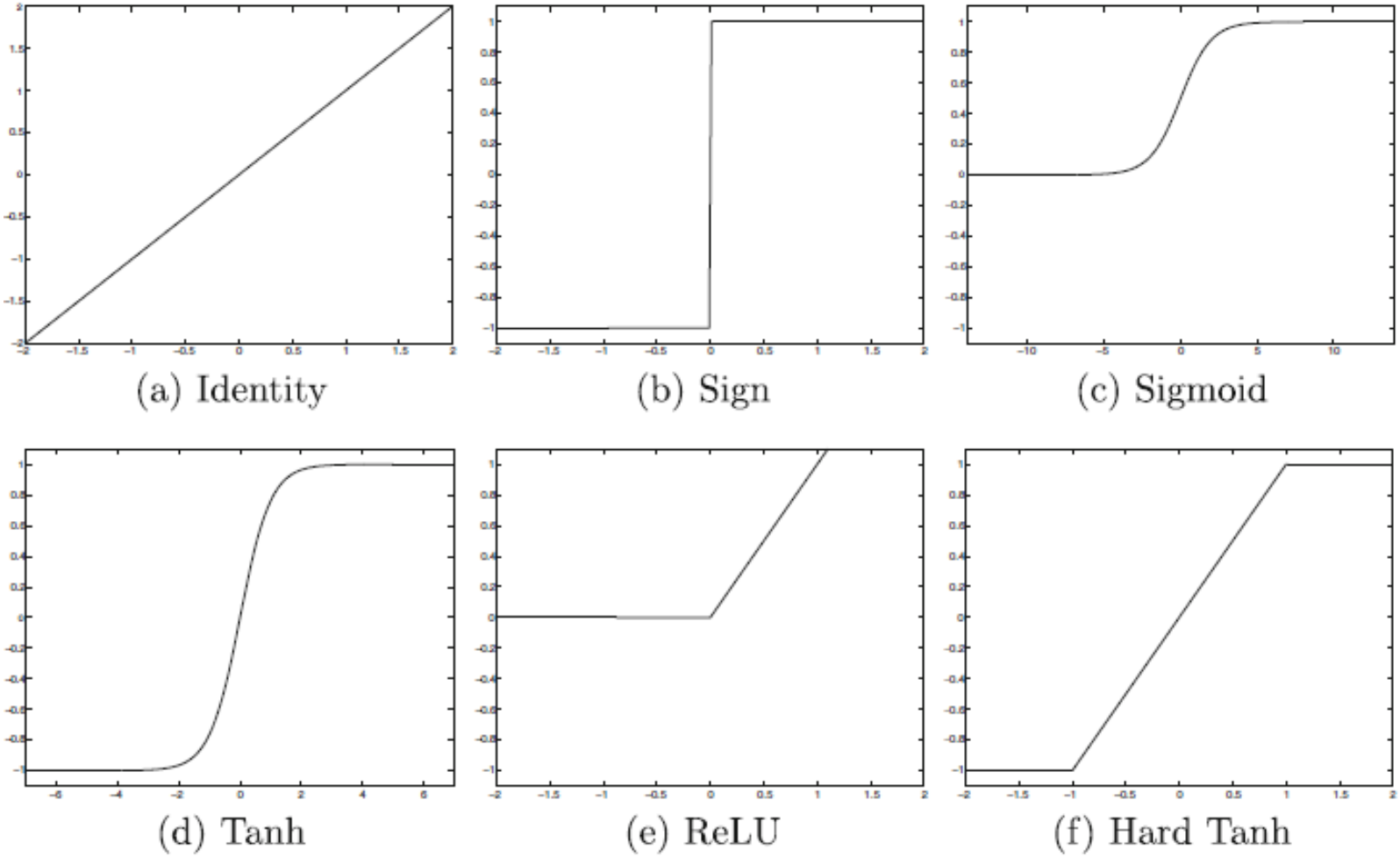
$$\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1} = \frac{e^{v}-e^{-v}}{e^{v}+e^{-v}} \quad (\tanh - \text{hyperbolic tangent})$$

$$\Phi(v) = \max\{v, 0\} \quad \text{(Rectified Linear Unit ReLU)}$$

$$\Phi(v) = \max\{\min\{1, v\}, -1\} \quad \text{(hard tanh)}$$

- Note that $\tanh(v) = 2\text{sigmoid}(2v) - 1$

Graphs of Activation Functions



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Traditional Uses of Activation

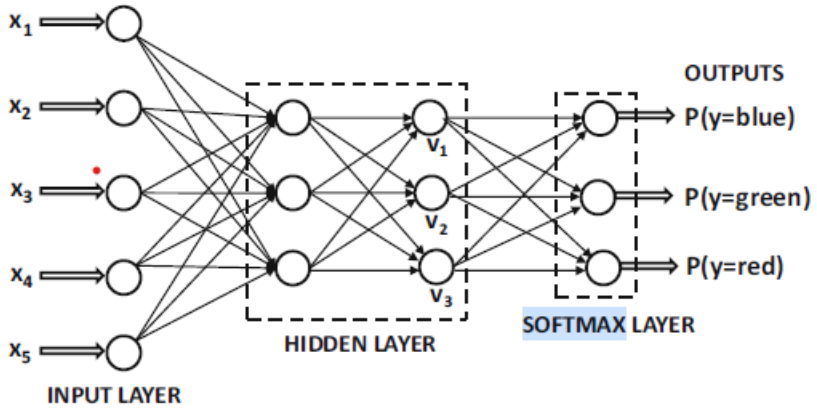
- Linear activation is often used when target (to predict) variable is real, and there is reason to believe that dependency on features is linear. The resulting algorithm is the same as least-squares regression (later)
- Sign activation can be used to map to binary outputs at prediction time. But its non-differentiability prevents its use for creating the loss function at training time. The perceptron criterion in training only works with linear activation (i.e. non trainable).
- The sigmoid activation outputs a value in $(0, 1)$, which can be interpreted as probabilities. It is possible to construct loss functions derived from maximum-likelihood models which allows for training

Traditional Uses of Activation (continued)

- $\tanh$ is similar to sigmoid, except that it is horizontally re-scaled and vertically translated to $[-1, 1]$ so it is better suited than sigmoid when the outputs must be both positive and negative. Its mean-centering and larger gradient steps make it easier to train
- Most popular are ReLU and hard tanh because of very good loss functions and hence the ease in training multilayered neural networks with these activation functions.

SoftMax: Special Case of Activation

- To represent a probability distribution of a class variable taking values $\{1, \dots, n\}$ the softmax activation is used.



- We must produce vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\vec{x})$ such that $\hat{y}_i \in \{0,1\}$ and $\sum_{i=1}^{n} \hat{y}_i = 1$ for $\hat{\boldsymbol{y}}$ to be a distribution. To do so
  1. Linear layer predicts unnormalized logs of probabilities:

$$z_i = \vec{w}_i \cdot \vec{x} + b_i$$

  2. The softmax activation then exponentiate and normalize $\vec{z}$ to obtain the desired $\hat{\boldsymbol{y}}$, so

$$\Phi(\vec{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

Typical Choice of Loss Function

Deterministic:

- Least-squares regression perceptron with numeric outputs requires a simple squared loss of the form $(y - \hat{y})^2$ for a single training instance with target y and prediction $\hat{y}$.
- Another type of loss for real valued prediction in 2-class classification is *hinge loss,* i.e. for $(\vec{x}, y)$ where $y \in \{-1, +1\}$ and real-valued prediction $\hat{y}$ (i.e. identity activation) hinge loss is $L(\vec{x}) = \max\{0, 1 - y \cdot \hat{y}\}$

Typical Choice of Loss Function (continued)

Probabilistic:

Binary targets:

- The observed value is assumed to be drawn from $\{-1, +1\}$, and the prediction $\hat{y}$ is any real on using the identity activation. In this case, the loss function for a single instance is:

$$L(x) = \log(1 + \exp(-y \cdot \hat{y}))$$

  aka *logistic regression*.

- For the same target can use sigmoid activation for prediction $\hat{y} \in (0, 1)$ of the same target. This way we model probability that observed value y is 1. Then loss is $L(x) = -\log\left|\frac{y}{2} - 0.5 + \hat{y}\right|$. This is because $\left|\frac{y}{2} - 0.5 + \hat{y}\right|$ indicates the probability that the prediction is correct.

Case-in-point: two different combinations of activation and loss functions can achieve the same result.

Typical Choice of Loss Function (continued)

Probabilistic:

- **Finite-valued Targets:** let $\hat{y}_1, \ldots, \hat{y}_k$ are the probabilities of the k classes using the softmax activation and the r^{th} class is the true class of an instance, then the loss function (for this single instance) is

$$L = -\log(\hat{y}_r)$$

  It implements multinomial logistic regression, and it is referred to as the cross-entropy loss.

Quick manipulation shows that binary logistic regression is identical to multinomial logistic regression, when the value of k is set to 2.

Typical Choice of Loss Function- Observations

- For discrete-valued outputs, it is common to use softmax activation with cross-entropy loss.
- For real-valued outputs, it is common to use linear activation with squared loss.
- Generally, cross-entropy loss is easier to optimize than squared loss.

Reading

- Ch. 1.1-1.2